SQL LAST TIME NOTES

SOURCE-SQL BOLT

A relational database represents a collection of related (two-dimensional) tables. Each of the tables are similar to an Excel spreadsheet, with a fixed number of named columns (the attributes or properties of the table) and any number of rows of data.

**SQL Lesson 1: SELECT queries 101**

Select query for a specific columns

SELECT column, another\_column, …

FROM mytable;

Select query for all columns

SELECT \*

FROM mytable;

**SQL Lesson 2: Queries with constraints (Pt. 1)**

in order to filter certain results from being returned, we need to use a **WHERE** clause in the query. The clause is applied to each row of data by checking specific column values to determine whether it should be included in the results or not.

Select query with constraints

SELECT column, another\_column, …

FROM mytable

**WHERE *condition***

**AND/OR *another\_condition***

**AND/OR …**;

More complex clauses can be constructed by joining numerous **AND** or **OR** logical keywords

|  |  |  |
| --- | --- | --- |
| **Operator** | **Condition** | **SQL Example** |
| =, !=, < <=, >, >= | Standard numerical operators | col\_name **!=** 4 |
| BETWEEN … AND … | Number is within range of two values (inclusive) | col\_name **BETWEEN** 1.5 **AND** 10.5 |
| NOT BETWEEN … AND … | Number is not within range of two values (inclusive) | col\_name **NOT BETWEEN** 1 **AND** 10 |
| IN (…) | Number exists in a list | col\_name **IN** (2, 4, 6) |
| NOT IN (…) | Number does not exist in a list | col\_name **NOT IN** (1, 3, 5) |

**SQL Lesson 3: Queries with constraints (Pt. 2)**

When writing **WHERE** clauses with columns containing text data, SQL supports a number of useful operators to do things like case-insensitive string comparison and wildcard pattern matching. We show a few common text-data specific operators below:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Condition** | **Example** |
| = | Case sensitive exact string comparison (**notice the single equals**) | col\_name **=** "abc" |
| != or <> | Case sensitive exact string inequality comparison | col\_name **!=** "abcd" |
| LIKE | Case insensitive exact string comparison | col\_name **LIKE** "ABC" |
| NOT LIKE | Case insensitive exact string inequality comparison | col\_name **NOT LIKE** "ABCD" |
| % | Used anywhere in a string to match a sequence of zero or more characters (only with LIKE or NOT LIKE) | col\_name **LIKE** "%AT%" (matches "AT", "ATTIC", "CAT" or even "BATS") |
| \_ | Used anywhere in a string to match a single character (only with LIKE or NOT LIKE) | col\_name **LIKE** "AN\_" (matches "AND", but not "AN") |
| IN (…) | String exists in a list | col\_name **IN** ("A", "B", "C") |
| NOT IN (…) | String does not exist in a list | col\_name **NOT IN** ("D", "E", "F") |

**Did you know?**

**All strings must be quoted so that the query parser can distinguish words in the string from SQL keywords.**

**SQL Lesson 4: Filtering and sorting Query results**

 SQL provides a convenient way to discard rows that have a duplicate column value by using the **DISTINCT** keyword.

Select query with unique results

SELECT **DISTINCT** column, another\_column, …

FROM mytable

WHERE *condition(s)*;

 SQL provides a way to sort your results by a given column in ascending or descending order using the **ORDER BY** clause.

When an **ORDER BY** clause is specified, each row is sorted alpha-numerically based on the specified column's value.

Another clause which is commonly used with the **ORDER BY** clause are the **LIMIT** and **OFFSET** clauses. The **LIMIT** will reduce the number of rows to return, and the optional **OFFSET** will specify where to begin counting the number rows from.

Select query with limited rows

SELECT column, another\_column, …

FROM mytable

WHERE *condition(s)*

ORDER BY column ASC/DESC

**LIMIT num\_limit OFFSET num\_offset**;

**SQL Lesson 5: Multi-table queries with JOINs**

Database normalization

Database normalization is useful because it minimizes duplicate data in any single table, and allows for data in the database to grow independently of each other (ie. Types of car engines can grow independent of each type of car).

# Multi-table queries with JOINs

Using the **JOIN** clause in a query, we can combine row data across two separate tables using this unique key. The first of the joins that we will introduce is the **INNER JOIN/JOIN.**

**INNER JOIN** might not be sufficient because the resulting table only contains data that belongs in both of the tables.

f the two tables have asymmetric data, , then we would have to use a **LEFT JOIN**, **RIGHT JOIN** or **FULL JOIN**

Select query with LEFT/RIGHT/FULL JOINs on multiple tables

SELECT column, another\_column, …

FROM mytable

**INNER/LEFT/RIGHT/FULL JOIN another\_table**

**ON mytable.id = another\_table.matching\_id**

WHERE *condition(s)*

ORDER BY column, … ASC/DESC

LIMIT num\_limit OFFSET num\_offset;

Like the **INNER JOIN** these three new joins have to specify which column to join the data on.  
When joining table A to table B, a **LEFT JOIN** simply includes rows from A regardless of whether a matching row is found in B. The **RIGHT JOIN** is the same, but reversed, keeping rows in B regardless of whether a match is found in A. Finally, a **FULL JOIN** simply means that rows from both tables are kept, regardless of whether a matching row exists in the other table.

**SQL Lesson 6: A short note on NULLs**

Sometimes, it's also not possible to avoid **NULL** values, . In these cases, you can test a column for **NULL** values in a **WHERE** clause by using either the **IS NULL** or **IS NOT NULL** constraint.

Select query with constraints on NULL values

SELECT column, another\_column, …

FROM mytable

**WHERE column IS/IS NOT NULL**

AND/OR *another\_condition*

AND/OR …;

**SQL Lesson 7: Queries with expressions**

Select query with expression aliases

SELECT ***col\_expression* AS *expr\_description***, …

FROM a**\_long\_table AS short\_table**;

**SQL Lesson 8: Queries with aggregates (Pt. 1)**

|  |  |
| --- | --- |
| **Function** | Description |
| **COUNT(**\***)**, **COUNT(***column***)** | A common function used to counts the number of rows in the group if no column name is specified. Otherwise, count the number of rows in the group with non-NULL values in the specified column. |
| **MIN(***column***)** | Finds the smallest numerical value in the specified column for all rows in the group. |
| **MAX(***column***)** | Finds the largest numerical value in the specified column for all rows in the group. |
| **AVG(***column*) | Finds the average numerical value in the specified column for all rows in the group. |
| **SUM(***column***)** | Finds the sum of all numerical values in the specified column for the rows in the group. |

Select query with aggregate functions over groups

SELECT AGG\_FUNC(*column\_or\_expression*) AS aggregate\_description, …

FROM mytable

WHERE *constraint\_expression*

**GROUP BY column**;

**SQL Lesson 9: Queries with aggregates (Pt. 2)**

One thing that you might have noticed is that if the **GROUP BY** clause is executed after the **WHERE** clause (which filters the rows which are to be grouped), then how exactly do we filter the grouped rows?

Luckily, SQL allows us to do this by adding an additional **HAVING** clause which is used specifically with the **GROUP BY** clause to allow us to filter grouped rows from the result set.

Select query with HAVING constraint

SELECT group\_by\_column, AGG\_FUNC(*column\_expression*) AS aggregate\_result\_alias, …

FROM mytable

WHERE *condition*

GROUP BY column

**HAVING *group\_condition***;

**Did you know?**

**If you aren't using the `GROUP BY` clause, a simple `WHERE` clause will suffice.**

**SQL Lesson 10: Inserting rows**

Insert statement with specific columns

INSERT INTO mytable

**(column, another\_column, …)**

VALUES (value\_or\_expr, another\_value\_or\_expr, …),

(value\_or\_expr\_2, another\_value\_or\_expr\_2, …),

…;

**SQL Lesson 11: Updating rows**

Update statement with values

UPDATE mytable

SET column = value\_or\_expr,

other\_column = another\_value\_or\_expr,

…

WHERE condition;

**SQL Lesson 12: Deleting rows**

When you need to delete data from a table in the database, you can use a **DELETE** statement, which describes the table to act on, and the rows of the table to delete through the **WHERE** clause. If you decide to leave out the **WHERE** constraint, then *all* rows are removed, which is a quick and easy way to clear out a table completely (if intentional).

Delete statement with condition

DELETE FROM mytable

WHERE condition;

# Taking care

**WHERE** clause (which causes the update to apply to *all* rows), you need to be extra careful when constructing **UPDATE** statements.

Without a proper backup or test database, it is downright easy to irrevocably remove data, so always read your **DELETE** statements twice and execute once.

**SQL Lesson 13: Creating tables**

Create table statement w/ optional table constraint and default value

CREATE TABLE IF NOT EXISTS mytable (

column *DataType* *TableConstraint* DEFAULT *default\_value*, another\_column *DataType* *TableConstraint* DEFAULT *default\_value*,

…

);

If there already exists a table with the same name, the SQL implmentation will usually throw an error, so to suppress the error and skip creating a table if one exists, you can use the **IF NOT EXISTS** clause.

# Table data types

|  |  |
| --- | --- |
| **Data type** | Description |
| **INTEGER**, **BOOLEAN** | The integer datatypes can store whole integer values like the count of a number or an age. In some implementations, the boolean value is just represented as an integer value of just 0 or 1. |
| **FLOAT**, **DOUBLE**, **REAL** | The floating point datatypes can store more precise numerical data like measurements or fractional values. Different types can be used depending on the floating point precision required for that value. |
| **CHARACTER(num\_chars)**, **VARCHAR(num\_chars)**, **TEXT** | The text based datatypes can store strings and text in all sorts of locales. The distinction between the various types generally amount to underlaying efficiency of the database when working with these columns.  Both the CHARACTER and VARCHAR (variable character) types are specified with the max number of characters that they can store (longer values may be truncated), so can be more efficient to store and query with big tables. |
| **DATE**, **DATETIME** | SQL can also store date and time stamps to keep track of time series and event data. They can be tricky to work with especially when manipulating data across timezones. |
| **BLOB** | Finally, SQL can store binary data in blobs right in the database. These values are often opaque to the database, so you usually have to store them with the right metadata to requery them. |

# Table constraints

|  |  |
| --- | --- |
| **Constraint** | Description |
| **PRIMARY KEY** | This means that the values in this column are unique, and each value can be used to identify a single row in this table. |
| **AUTOINCREMENT** | For integer values, this means that the value is automatically filled in and incremented with each row insertion. Not supported in all databases. |
| **UNIQUE** | This means that the values in this column have to be unique, so you can't insert another row with the same value in this column as another row in the table. Differs from the `PRIMARY KEY` in that it doesn't have to be a key for a row in the table. |
| **NOT NULL** | This means that the inserted value can not be `NULL`. |
| **CHECK (expression)** | This is allows you to run a more complex expression to test whether the values inserted are value. For example, you can check that values are positive, or greater than a specific size, or start with a certain prefix, etc. |
| **FOREIGN KEY** | This is a consistency check which ensures that each value in this column corresponds to another value in a column in another table.  For example, if there are two tables, one listing all Employees by ID, and another listing their payroll information, the `FOREIGN KEY` can ensure that every row in the payroll table corresponds to a valid employee in the master Employee list. |

An example

Movies table schema

CREATE TABLE movies (

id INTEGER PRIMARY KEY,

title TEXT,

director TEXT,

year INTEGER,

length\_minutes INTEGER

);

**SQL Lesson 14: Altering tables**

Adding columns

Altering table to add new column(s)

ALTER TABLE mytable

ADD column *DataType* *OptionalTableConstraint*

DEFAULT default\_value;

Removing columns

Dropping columns is as easy as specifying the column to drop

Altering table to remove column(s)

ALTER TABLE mytable

DROP column\_to\_be\_deleted;

Renaming the table

Altering table name

ALTER TABLE mytable

RENAME TO new\_table\_name;

**SQL Lesson 15: Dropping tables**

In some rare cases, you may want to remove an entire table including all of its data and metadata, and to do so, you can use the **DROP TABLE** statement, which differs from the **DELETE** statement in that it also removes the table schema from the database entirely.

Drop table statement

DROP TABLE IF EXISTS mytable;

* Like the **CREATE TABLE** statement, the database may throw an error if the specified table does not exist, and to suppress that error, you can use the **IF EXISTS** clause.
* In addition, if you have another table that is dependent on columns in table you are removing (for example, with a **FOREIGN KEY** dependency) then you will have to either update all dependent tables first to remove the dependent rows or to remove those tables entirely.

**SQL Lesson 16: Order of execution of a Query**

Complete SELECT query

SELECT DISTINCT column, AGG\_FUNC(*column\_or\_expression*), …

FROM mytable

JOIN another\_table

ON mytable.column = another\_table.column

WHERE *constraint\_expression*

GROUP BY column

HAVING *constraint\_expression*

ORDER BY *column* ASC/DESC

LIMIT *count* OFFSET *COUNT*;

## 1.FROM and JOINs

## 2.  WHERE

## 3. GROUP BY- you should only need to use this when you have aggregate functions in your query.

## 4. HAVING

## 5. SELECT

## 6. DISTINCT

## 7. ORDER BY

## 8. LIMIT / OFFSET